DLP Lab2 Report

Shu Kai, Lin

March 2025

1 Implementation Details

1.1 Training

```python
seed = 42
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
if torch.cuda.is_available():
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

def train(args, model, train_loader, valid_loader):
    losses, dices = [], []
    device = torch.device(f"cuda:{args.cuda}" if torch.cuda.is_available() else "cpu")
    if args.optim == "Adam":
        optimizer = optim.Adam(model.parameters(), lr=args.learning_rate)
    elif args.optim == "AdamW":
        optimizer = optim.AdamW(model.parameters(), lr=args.learning_rate)
    else:
        optimizer = optim.SGD(model.parameters(), lr=args.learning_rate, momentum=0.9)
    if agrs.scheduler == "Cos":
        scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=args.epoches,
        ↪  eta_min=args.lr_eta)
    else:
        scheduler = optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.95)
    if args.loss_func == "BCE":
        critirion = nn.BCEWithLogitsLoss()
    else:
        critirion = BCEDiceLoss(weight_bce=0.5, weight_dice=0.5, use_logits=True)
    scaler = GradScaler()  # Helps stabilize float16 training
    save_dir = os.path.join(args.root,"saved_models/")
    os.makedirs(save_dir, exist_ok=True)  # Ensure directory exists
    checkpoint_path = os.path.join(save_dir, f"{args.model}_best_model.pth")
    best_dice = 0.0
    for ep in range(args.epoches):
        train_loss = 0.0
        model.train()
        progress_bar = tqdm(iter(train_loader), desc=f"Epoch {ep+1}/{args.epoches}", leave=True)
        for img, mask in progress_bar:
            img, mask = img.to(device), mask.to(device)
            optimizer.zero_grad()
            with autocast():
                pred_mask = model(img)  # Ensure model outputs (N, C, H, Wd)
                loss = critirion(pred_mask, mask)  # Compute loss
            scaler.scale(loss).backward()
            scaler.step(optimizer)
            scaler.update()
            train_loss += loss.item()
            progress_bar.set_postfix(loss=f"{loss.item():.4f}")
        train_loss /= len(train_loader)
        model.eval()
        with torch.no_grad():
            sum_dice = 0
            for img, mask in tqdm(valid_loader, desc="Validating", leave=False):
                img, mask = img.to(device), mask.to(device)

                with autocast():
                    pred_mask = model(img)
                pred_mask = torch.sigmoid(pred_mask)
                sum_dice += dice_score(pred_mask, mask)
            dice = sum_dice / len(valid_loader)
        if dice > best_dice:
            best_dice = dice
            torch.save({
                'epoch': ep + 1,
                'model_state_dict': model.state_dict(),
                'optimizer_state_dict': optimizer.state_dict(),
                'dice_score': best_dice
            }, checkpoint_path)
            print(f" Saved Best Model at {checkpoint_path} (Epoch {ep+1}, Dice Score:
            ↪  {best_dice:.4f})")
        print(f"Epoch {ep+1}, Loss: {train_loss:.4f}, Dice Score: {dice:.4f}, LR:
        ↪  {scheduler.get_last_lr()[0]:.6f}")
        scheduler.step()
        losses.append(train_loss)
        dices.append(dice)
    with open(f"./saved_data/{args.model}_{args.loss_func}_loss.cmp.txt","a") as f:
        f.write(f"losses :")
        output_str = "\n".join(str(item) for item in losses)
        f.write(output_str)
    with open(f"./saved_data/{args.model}_{args.loss_func}_dice.cmp.txt","a") as f:
        f.write(f"dices :")
        output_str = "\n".join(str(item) for item in dices)
        f.write(output_str)
    save_dir = os.path.join("./saved_fig",f"{args.model}_{args.loss_func}_
                            {args.learning_rate}_loss_dice.png")
    plot_metrics(save_dir = save_dir,epochs= args.epoches,
                 loss_values=losses, dice_scores= dices)
if __name__ == "__main__":
    args = parse_arguments()
    device = torch.device(f"cuda:{args.cuda}" if torch.cuda.is_available() else "cpu")
    if args.model == "unet":
        print("Using unet")
        model = UNet(3,1).to(device)
    else:
        print("Using resnet34_unet")
```

```python
        model = ResNet34_UNet(3,1).to(device)
    if args.load_pt != "":
        ckpt_dir = os.path.join(args.root,"saved_models/",args.load_pt)
    else:
        ckpt_dir = os.path.join(args.root,"saved_models/",f"{args.model}_best_model.pth")
    try:
        checkpoint = torch.load(ckpt_dir)
        model.load_state_dict(checkpoint['model_state_dict'])
        print(f" Loading model from {ckpt_dir}'")
    except Exception as e:
        print(f" Can't load model from {ckpt_dir}, retype the correct relative path")
        raise RuntimeError(f"Model loading failed from {ckpt_dir}") from e
    data_module = OxfordPetData(root_dir=args.root, batch_size=args.batch, num_workers=8)
    _, _, test_loader = data_module.get_dataloaders()
    print("####Testing####")
    dice = evaluate(device=device, model=model, test_loader=test_loader)
    print(f"Dice score : {dice:.4f}")
```

For the training component, I first stabilized all random seeds to ensure that each run produces consistent results. During training, I used the AdamW optimizer, which requires less hyperparameter tuning and maintains a better balance between parameter updates and regularization compared to Adam.
The learning rate scheduler, CosineAnnealingLR, gradually reduces the learning rate using a cosine function for a smoother decay. In addition to binary cross-entropy, I implemented an extra loss function that combines Dice loss, aligning better with the task requirements. Since the model does not apply an activation function in the output layer, the loss function operates on logits to remain compatible with FP16 precision.
To speed up training, I utilized FP16 precision, as it achieved nearly the same accuracy as FP32 in my tests. In every epoch, I computed both the loss and Dice score, updating the final checkpoint with the best Dice score.

1.2 Dice Score

```python
def dice_score(predicted: torch.Tensor, ground_truth: torch.Tensor, eps: float = 1e-6) -> float:
    with torch.no_grad():
        predicted = predicted.float()
        ground_truth = ground_truth.float()
        intersection = torch.sum(predicted * ground_truth)
        sum_sizes = torch.sum(predicted) + torch.sum(ground_truth)
        dice = (2.0 * intersection + eps) / (sum_sizes + eps)  # Add epsilon to avoid division by zero
        return dice.item()
```

Calculating dice score within batches and implementaion of dice score formula. An extra epsilon constant is added to avoid divison by zero trap.

1.3 BCE with Dice loss

```python
class DiceLoss(nn.Module):
    def __init__(self, smooth: float = 1e-6):
        super(DiceLoss, self).__init__()
        self.smooth = smooth

    def forward(self, inputs: torch.Tensor, targets: torch.Tensor) -> torch.Tensor:
        inputs = inputs.contiguous().view(-1)
        targets = targets.contiguous().view(-1)
        intersection = (inputs * targets).sum()
        dice = (2.0 * intersection + self.smooth) / (inputs.sum() + targets.sum() + self.smooth)
        return 1 - dice

class BCEDiceLoss(nn.Module):
    def __init__(self, weight_bce: float = 0.5, weight_dice: float = 0.5, use_logits: bool = True):
        super(BCEDiceLoss, self).__init__()
        self.weight_bce = weight_bce
        self.weight_dice = weight_dice
        self.use_logits = use_logits

        if self.use_logits:
            self.bce = nn.BCEWithLogitsLoss()
        else:
            self.bce = nn.BCELoss()
        self.dice = DiceLoss()
    def forward(self, inputs: torch.Tensor, targets: torch.Tensor) -> torch.Tensor:
        loss_bce = self.bce(inputs, targets)
        if self.use_logits:
            inputs = torch.sigmoid(inputs)
        loss_dice = self.dice(inputs, targets)
        return self.weight_bce * loss_bce + self.weight_dice * loss_dice
```

The function first computes the Dice Loss, which is defined as 1 minus the Dice Coefficient. It then calculates the binary cross-entropy (BCE) loss separately. Finally, the two losses are combined—typically using a weighted sum—to form the overall loss optimized during training.

1.4 Plot Prediction

```
def plot_metrics(save_dir, epochs, loss_values, dice_scores):
    plt.figure(figsize=(10, 5))
    epoch_range = list(range(1, epochs + 1))
    plt.plot(epoch_range, loss_values, label="Loss", color="blue")
    plt.plot(epoch_range, dice_scores, label="Dice Score", color="orange")
    plt.xlabel("Epoch")
    plt.ylabel("Value")
    plt.title("Training Loss and Dice Score Over Epochs")
    plt.legend()
    plt.grid(True)
    output_dir = os.path.dirname(save_dir)
    if not os.path.exists(output_dir):
        print(f"Directory '{output_dir}' does not exist. Creating it now...")
        os.makedirs(output_dir)
    plt.savefig(save_dir, bbox_inches="tight")
    plt.close()
    # plt.show()
```

Plotting the prediction mask and concating with the inputs.

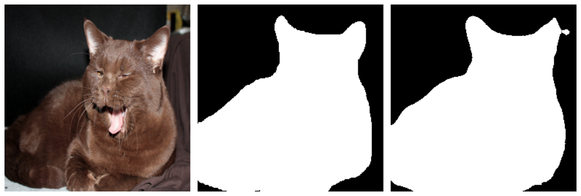


Figure 1: Result Example

1.5 Evaluate

```python
def evaluate(device, model, test_loader):
    model.to(device)
    model.eval()
    with torch.no_grad():
        sum = 0
        for i, (img, mask) in tqdm(enumerate(test_loader), total=len(test_loader)):
            img, mask = img.to(device), mask.to(device)
            pred_mask = model(img)
            pred_mask = torch.sigmoid(pred_mask)
            pred_mask = pred_mask > 0.4 #best 0.4
            sum += dice_score(pred_mask, mask)
            if i % 10 == 0:
                plot_result(i, img=img, mask=mask, pred_mask=pred_mask)
    return sum / len(test_loader)
```

This function evaluates the Dice score during the testing phase. For the predicted mask values, it will go through a sigmoid activation function to ensure its bounded between 0 -1 and a threshold of 0.4 is applied since it provides the best results. Additionally, every 10 iterations, the function calls plot_result to visualize the results for further analysis.

1.6 Inference

```python
seed = 42
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
if torch.cuda.is_available():
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
if __name__ == "__main__":
    args = parse_arguments()
    device = torch.device(f"cuda:{args.cuda}" if torch.cuda.is_available() else "cpu")
    if args.model == "unet":
        print("Using unet")
        model = UNet(3,1).to(device)
    else:
        print("Using resnet34_unet")
        model = ResNet34_UNet(3,1).to(device)
    if args.load_pt != "":
        ckpt_dir = os.path.join(args.root,"saved_models/",args.load_pt)
    else:
        ckpt_dir = os.path.join(args.root,"saved_models/",f"{args.model}_best_model.pth")
    try:
        checkpoint = torch.load(ckpt_dir)
        model.load_state_dict(checkpoint['model_state_dict'])
        print(f" Loading model from {ckpt_dir}'")
    except:
        print(f" Can't load model from {ckpt_dir}, retype correct relative path")

    data_module = OxfordPetData(root_dir=args.root, batch_size=args.batch, num_workers=8)
    _, _, test_loader = data_module.get_dataloaders()
    print("####Testing####")
    dice = evaluate(device=device, model=model, test_loader=test_loader)
    print(f"Dice score : {dice:.4f}")
```

For the inference stage, the checkpoint is loaded from the provided directory; if none is given, the best checkpoint for the target model is loaded by default. Then, the test dataloader is obtained and passed to the evaluation function, which computes the dice scores for testing and plots the predicted masks. Also, random seed is also stablized.

1.7 Unet

[1]

```
class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ConvBlock, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )
    def forward(self, x):
        return self.conv(x)
class EncoderBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(EncoderBlock, self).__init__()
        self.conv_block = ConvBlock(in_channels, out_channels)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
    def forward(self, x):
        features = self.conv_block(x)
        pooled = self.pool(features)
        return pooled, features
class DecoderBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DecoderBlock, self).__init__()
        self.upsample = nn.ConvTranspose2d(in_channels, out_channels, kernel_size=2, stride=2)
        self.conv_block = ConvBlock(out_channels * 2, out_channels)

    def forward(self, x, skip):
        x = self.upsample(x)
        if x.size()[2:] != skip.size()[2:]:
            diff_y = skip.size(2) - x.size(2)
            diff_x = skip.size(3) - x.size(3)
            skip = skip[:, :, diff_y // 2: diff_y // 2 + x.size(2),
                          diff_x // 2: diff_x // 2 + x.size(3)]
        x = torch.cat([skip, x], dim=1)
        x = self.conv_block(x)
        return x
class UNet(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(UNet, self).__init__()
        self.enc1 = EncoderBlock(in_channels, 64)
        self.enc2 = EncoderBlock(64, 128)
        self.enc3 = EncoderBlock(128, 256)
        self.enc4 = EncoderBlock(256, 512)
        self.center = ConvBlock(512, 1024)
        self.dec4 = DecoderBlock(1024, 512)
        self.dec3 = DecoderBlock(512, 256)
        self.dec2 = DecoderBlock(256, 128)
        self.dec1 = DecoderBlock(128, 64)
        self.final_conv = nn.Conv2d(64, out_channels, kernel_size=1)

    def forward(self, x):
        enc1_pool, skip1 = self.enc1(x)
        enc2_pool, skip2 = self.enc2(enc1_pool)
        enc3_pool, skip3 = self.enc3(enc2_pool)
        enc4_pool, skip4 = self.enc4(enc3_pool)
        center = self.center(enc4_pool)
        dec4 = self.dec4(center, skip4)
        dec3 = self.dec3(dec4, skip3)
        dec2 = self.dec2(dec3, skip2)
        dec1 = self.dec1(dec2, skip1)
        out = self.final_conv(dec1)
        return out
```

My implementation of Unet architecture is referenced from:
https://github.com/milesial/Pytorch-UNet.git

The UNet architecture consists of an encoder (downsampling path), a central bottleneck block, and a decoder (upsampling path). The encoder extracts hierarchical features through ConvBlocks, each followed by 2×2 max pooling, progressively doubling the number of channels while halving spatial dimensions:

input(3, 256, 256) → encoder 1 → c1(64, 256, 256) → encoder 2 → c2(128, 128, 128) → encoder 3 → c3(256, 64, 64) → encoder 4 → c4(512, 32, 32).

Each EncoderBlock consists of a ConvBlock, which applies two 3×3 convolutions with BatchNorm and ReLU, followed by a 2×2 max pooling operation. At the central block, ConvBlock(512,1024) refines the deepest features without further downsampling before upsampling begins. The decoder path gradually restores spatial resolution by merging upsampled features with skip connections from the encoder. Each DecoderBlock first upsamples the feature map using a transposed convolution, concatenates it with its encoder counterpart, then applies ConvBlock(input_channel, input_channel/2), progressively reducing the number of channels:

c4(512, 32, 32) $\rightarrow$ center block $\rightarrow$ (1024, 32, 32) $\rightarrow$ decoder 4 $\rightarrow$ (512, 32, 32) + c4 $\rightarrow$ (512, 32, 32) $\rightarrow$ decoder 3 $\rightarrow$ (256, 64, 64) + c3 $\rightarrow$ (512, 64, 64) $\rightarrow$ decoder 2 $\rightarrow$ (128, 128, 128) + c2 $\rightarrow$ (256, 128, 128) $\rightarrow$ decoder 1 $\rightarrow$ (64, 256, 256) + c1 $\rightarrow$ (128, 256, 256).

Finally, the final convolution layer (conv(128,1)) maps the output to a single-channel segmentation mask, restoring the full resolution:

decoder 1 $\rightarrow$ output mask(1, 256, 256).

The encoder progressively extracts features while reducing spatial resolution, the central block refines high-level representations, and the decoder reconstructs the original resolution while integrating fine-grained details from skip connections. This design ensures precise localization, efficient feature extraction, and multi-scale information fusion. The implementation follows this structured approach, with EncoderBlock performing downsampling using ConvBlock and max pooling, DecoderBlock applying transposed convolutions and merging skip connections, and ConvBlock maintaining consistency in feature extraction. The UNet class combines four encoder blocks, a central bottleneck, four decoder blocks, and a final 1×1 convolution. The forward pass follows the structured encoder-decoder pattern: each encoder block outputs both pooled features (for deeper layers) and skip connections (for the decoder), while each decoder block restores spatial details before the final segmentation mask is generated.

1.8 Resnet34 with Unet

```python
class ResNetBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride):
        super(ResNetBlock, self).__init__()
        self.block = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride,
                      padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1,
                      padding=1, bias=False),
            nn.BatchNorm2d(out_channels)
        )
        self.down = None
        if stride != 1 or in_channels != out_channels:
            self.down = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride,
                          bias=False),
                nn.BatchNorm2d(out_channels)
            )
        self.relu = nn.ReLU(inplace=True)
    def forward(self, x):
        identity = x
        out = self.block(x)
        if self.down is not None:
            identity = self.down(x)
        out += identity
        out = self.relu(out)
        return out
class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ConvBlock, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3,
                      padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3,
                      padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        return self.conv(x)
class EncoderBlock(nn.Module):
    def __init__(self, in_channels, out_channels, n_blocks):
        super(EncoderBlock, self).__init__()
        layers = []
        layers.append(ResNetBlock(in_channels, out_channels, stride=2))
        for _ in range(1, n_blocks):
            layers.append(ResNetBlock(out_channels, out_channels, stride=1))
        self.block = nn.Sequential(*layers)
    def forward(self, x):
        skip = x
        out = self.block(x)
        return out, skip
class DecoderBlock(nn.Module):
    def __init__(self, in_channels, skip_channels, out_channels):
        super(DecoderBlock, self).__init__()
        self.up = nn.ConvTranspose2d(in_channels, out_channels,
                                     kernel_size=2, stride=2)
        self.conv = ConvBlock(out_channels + skip_channels, out_channels)
    def forward(self, x, skip):
        x = self.up(x)
        if x.size()[2:] != skip.size()[2:]:
            x = F.interpolate(x, size=skip.size()[2:], mode='bilinear',
                              align_corners=True)
        x = torch.cat([skip, x], dim=1)
        x = self.conv(x)
        return x
class ResNet34_UNet(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ResNet34_UNet, self).__init__()
        self.encoder1 = nn.Sequential(
            nn.Conv2d(in_channels, 64, kernel_size=7, stride=2, padding=3,
                      bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        )
        self.encoder2 = EncoderBlock(64, 64, n_blocks=3)
        self.encoder3 = EncoderBlock(64, 128, n_blocks=4)
        self.encoder4 = EncoderBlock(128, 256, n_blocks=6)
        self.encoder5 = EncoderBlock(256, 512, n_blocks=3)
        self.center = nn.Sequential(
            nn.Conv2d(512, 256, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True)
        )
        self.decoder4 = DecoderBlock(in_channels=256, skip_channels=256, out_channels=128)
        self.decoder3 = DecoderBlock(in_channels=128, skip_channels=128, out_channels=64)
        self.decoder2 = DecoderBlock(in_channels=64, skip_channels=64, out_channels=32)
        self.decoder1 = DecoderBlock(in_channels=32, skip_channels=64, out_channels=32)
        self.output = nn.Sequential(
            nn.ConvTranspose2d(32, 32, kernel_size=2, stride=2),
            ConvBlock(32, 32),
            nn.ConvTranspose2d(32, 32, kernel_size=2, stride=2),
            ConvBlock(32, 32),
            nn.Conv2d(32, out_channels, kernel_size=1)
        )

    def forward(self, x):
        enc1 = self.encoder1(x)
        enc2, skip1 = self.encoder2(enc1)
        enc3, skip2 = self.encoder3(enc2)
        enc4, skip3 = self.encoder4(enc3)
        enc5, skip4 = self.encoder5(enc4)
        center = self.center(enc5)
        dec4 = self.decoder4(center, skip4)
        dec3 = self.decoder3(dec4, skip3)
        dec2 = self.decoder2(dec3, skip2)
        dec1 = self.decoder1(dec2, skip1)

        return self.output(dec1)

```

My implementation of Resnet34 architecture is referenced from:
https://github.com/pytorch/vision/blob/main/torchvision/models/resnet.py

The ResNet34-UNet architecture is structured with an encoder-decoder framework where the encoder path begins with a 7×7 convolutional stem followed by batch normalization, ReLU activation, and max pooling before passing through four ResNet-based residual blocks, each consisting of a downsampling convolution with stride 2 followed by multiple residual layers that maintain spatial consistency while increasing feature depth, transforming the input as

input(3, 256, 256) → initial (64, 128, 128) → encoder1 (64, 128, 128) → encoder2 (128, 64, 64) → encoder3 (256, 32, 32) → encoder4 (512, 16, 16).

The residual blocks are implemented using two 3×3 convolutions with batch normalization and ReLU activations, with identity shortcuts for feature propagation, and include projection layers when input and output channels differ. The central block, which acts as a bottleneck, applies additional convolutions to refine features, increasing the channel count from 512 to 1024 without further downsampling. The decoder path restores spatial resolution through transposed convolutions, where each upsampling step doubles spatial dimensions and halves the number of channels, before concatenating with the corresponding skip connection from the encoder to retain fine-grained details, progressing as center (1024, 16, 16) + encoder4 → decoder4 (256, 32, 32) + encoder3 → decoder3 (128, 64, 64) + encoder2 → decoder2 (64, 128, 128) + encoder1 → decoder1 (64, 256, 256) → final output (1, 256, 256).

Each decoder block consists of a transposed convolution for upsampling, followed by a concatenation operation along the channel axis, and a convolutional refinement block that applies two 3×3 convolutions with batch normalization and ReLU to smooth and adjust feature representations. To ensure spatial alignment when concatenating skip connections, bilinear interpolation is used when necessary to adjust mismatched feature sizes. The final output is produced by a 1×1 convolution, mapping the 64-channel output of the last decoder block to the required number of output channels, ensuring that the architecture preserves spatial hierarchies, maintains feature consistency through residual connections, and efficiently reconstructs feature maps at multiple scales.

2 Data Preprocessing

2.1 Read Image & Mask

```python
class Read_data(Dataset):
    def __init__(self, root, list_file, H=256, W=256, indices=None, is_test = False):
        self.H = H
        self.W = W
        self.data_path = root
        df = pd.read_csv(list_file, sep=" ", header=None)
        names = df[0].values
        self.image_paths = [os.path.join(self.data_path, f"images/{name}.jpg") for name in names]
        self.mask_paths = [os.path.join(self.data_path, f"annotations/trimaps/{name}.png") for name
        in names]
        self.is_test = is_test
        if indices is not None:
            self.image_paths = [self.image_paths[i] for i in indices]
            self.mask_paths = [self.mask_paths[i] for i in indices]
    def __len__(self):
        return len(self.image_paths)
    def __getitem__(self, idx):
        img_path = self.image_paths[idx]
        mask_path = self.mask_paths[idx]
        try:
            image = Image.open(img_path).convert("RGB")
            image = image.resize((self.W, self.H))
            mask = Image.open(mask_path).convert("L")
            mask = mask.resize((self.W, self.H), resample=Image.NEAREST)
            if self.is_test == False:
                image, mask = joint_transform(image=image, mask=mask)
            else:
                image = TF.to_tensor(image)
                mask = TF.to_tensor(mask)
            mask = torch.where(mask == 1.0/255.0, 1.0, mask)
            mask = torch.where(mask == 3.0/255.0, 1.0, mask)
            mask = torch.where(mask == 2.0/255.0, 0.0, mask))
            return image, mask
        except Exception as e:
            print(f"Skipping corrupted image: {img_path}, Error: {e}")
            return None, None
```

For the data reading part, I wrote a custom Torch dataset class to handle the dataset. During initialization, this class takes the root directory of dataset/oxford-iit-pet/, the height and width for the images and masks, the index type, and an is_test flag as input. The paths for the image and mask data are managed using pandas, which reads a CSV file containing the directories of all images and masks. The indices help determine which images and masks belong to the training or validation sets.
In the __getitem__ function, which is used by the dataloader, the image and mask are opened and read with PIL and then resized to the specified height and width. Since only the training dataset undergoes augmentation, I pass an is_test flag to indicate whether the current data loader is for testing. If it is for testing, the data is directly transformed into tensors without augmentation. Additionally, the mask values are handled differently because the read-in mask values are not binary. I transform the mask values by mapping 1.0/255.0 to 0.0 and all other values to 1.0, which better aligns with the task requirements.

2.2 DataLoader

```
class OxfordPetData:
    def __init__(self, root_dir, batch_size=8, num_workers=8, H=256, W=256):
        self.root_dir = os.path.join(root_dir,"dataset/oxford-iiit-pet/")
        self.batch_size = batch_size
        self.num_workers = num_workers
        self.H = H
        self.W = W
        self._prepare_data()

    def _prepare_data(self):
        trainval_file = os.path.join(self.root_dir, "annotations/trainval.txt")
        test_file = os.path.join(self.root_dir, "annotations/test.txt")
        full_dataset = Read_data(self.root_dir, trainval_file, H=self.H, W=self.W)
        indices = list(range(len(full_dataset)))
        train_idx, valid_idx = train_test_split(indices, test_size=0.2, random_state=42)
        self.train_dataset = Read_data(self.root_dir, trainval_file, H=self.H, W=self.W,
        indices=train_idx, is_test=False)
        self.valid_dataset = Read_data(self.root_dir, trainval_file, H=self.H, W=self.W,
        indices=valid_idx, is_test=False)
        self.test_dataset = Read_data(self.root_dir, test_file, H=self.H, W=self.W,is_test=True)

    def get_dataloaders(self):
        train_loader = DataLoader(self.train_dataset, batch_size=self.batch_size, shuffle=True,
        ↪  num_workers=self.num_workers,pin_memory=True)
        valid_loader = DataLoader(self.valid_dataset, batch_size=self.batch_size, shuffle=False,
        ↪  num_workers=self.num_workers,pin_memory=True)
        test_loader = DataLoader(self.test_dataset, batch_size=self.batch_size, shuffle=False,
        ↪  num_workers=self.num_workers,pin_memory=True)
        return train_loader, valid_loader, test_loader

```

For the dataloader component, this section primarily manages the input variables for the dataset and dataloader. The key function is _prepare_data, which handles the train, validation, and test file paths by locating the corresponding txt files and splitting the training and validation sets using sklearn's built-in function. The get_dataloader function then returns all the dataloaders.

2.3 Image augumentation

```python
def joint_transform(image, mask, degrees=15, flip_prob=0.5):
    angle = random.uniform(-degrees, degrees)
    rotated_image = TF.rotate(image, angle)
    rotated_mask = TF.rotate(mask, angle)
    if random.random() < flip_prob:
        rotated_image = TF.hflip(rotated_image)
        rotated_mask = TF.hflip(rotated_mask)
    image_tensor = TF.to_tensor(rotated_image)
    mask_tensor = TF.to_tensor(rotated_mask)
    return image_tensor, mask_tensor
```

For the image augmentation part, since I load all images and masks from paths using PIL, I can't use torchvision's built-in functions for manipulation. Instead, I wrote a helper function that augments the image and mask simultaneously. The function takes an image, a mask, a rotation bound, and a flipping probability as inputs. A random rotation angle is generated, and both the image and mask are rotated by the same angle. For the horizontal flip, the decision to flip is made based on the provided flipping probability. Finally, the image and mask are converted into tensors and returned.

2.4 Why Method is Unique

My method mainly applies horizontal flips and rotations within ±15 degrees to add diversity to the training dataset. Through extensive trial and error, I eventually achieved 93% accuracy using ResNet. Therefore, I believe that my augmentation approach is highly successful.

3 Analyze the experiment results

3.1 Different Model Architecture

With hyperparameters as follow: learning rate = 1e-4, Loss function using Binary Cross entropy, scheduler using cosineAnnealinear decaying learning rate to 1e-5, optimizer using AdamW

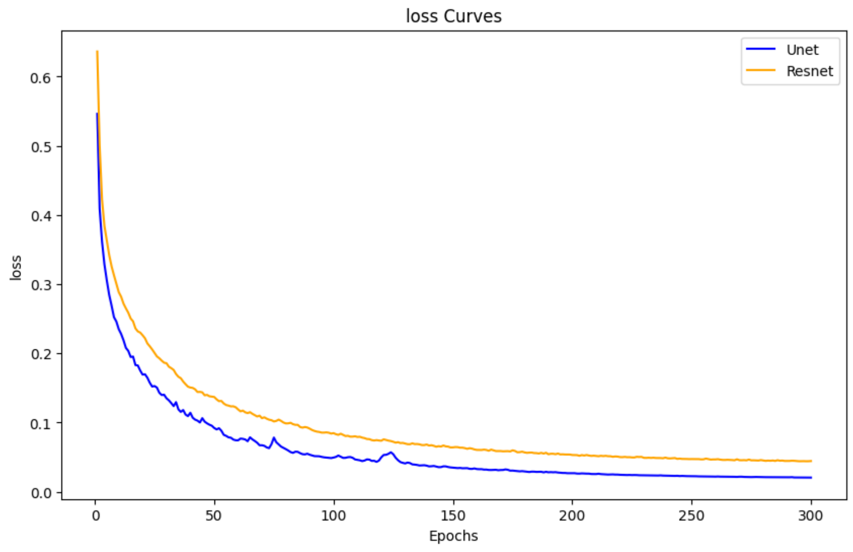


Figure 2: Unet VS Resnet34 Loss comparsion

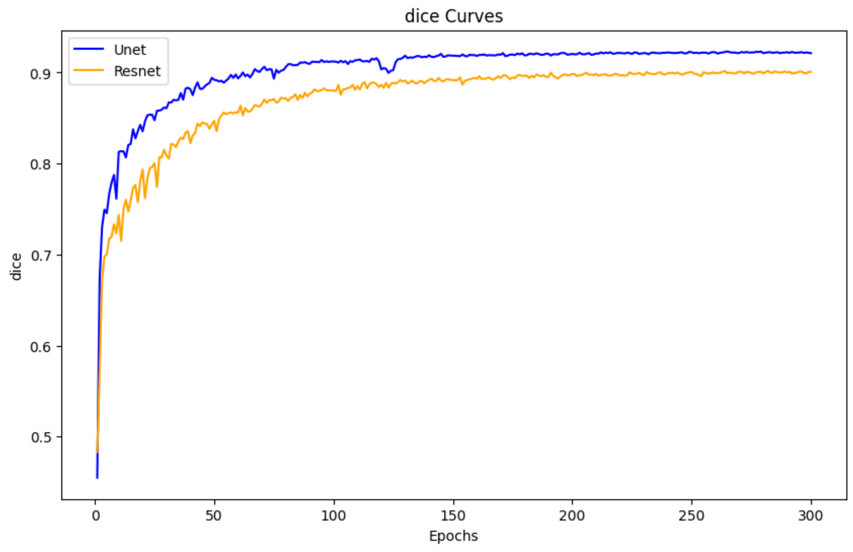
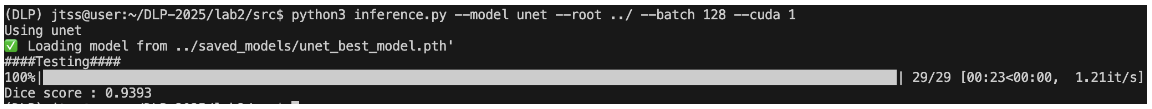
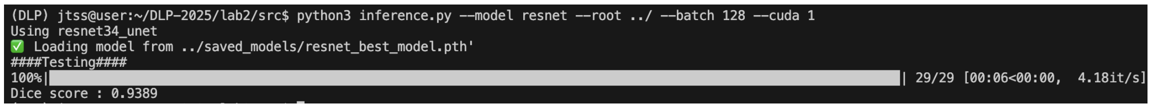


Figure 3: Unet VS Resnet34 Dice Score comparsion



```
(DLP) jtss@user:~/DLP-2025/lab2/src$ python3 inference.py --model unet --root ../ --batch 128 --cuda 1
Using unet
✅ Loading model from ../saved_models/unet_best_model.pth'
####Testing####
100%|████████████████████████████████████████████████████████| 29/29 [00:23<00:00,  1.21it/s]
Dice score : 0.9393
```

Figure 4: Unet Testing Outcome



```
(DLP) jtss@user:~/DLP-2025/lab2/src$ python3 inference.py --model resnet --root ../ --batch 128 --cuda 1
Using resnet34_unet
✅ Loading model from ../saved_models/resnet_best_model.pth'
####Testing####
100%|████████████████████████████████████████████████████████| 29/29 [00:06<00:00,  4.18it/s]
Dice score : 0.9389
```

Figure 5: Resnet34 Testing Outcome

From the figures above can discover that since Unet has a more complicated encoder and more paratmeters to memorize weight which can lead to higer accuracy. However, this will lead to a tradeoff that the training time of Unet will be 4 times more than Resnet34(Unet: 2hr, Resnet34: 30 min).Yet as the accruracy is actually quite close, I think it isn't a great trade off.

3.2 Different Learning Rate

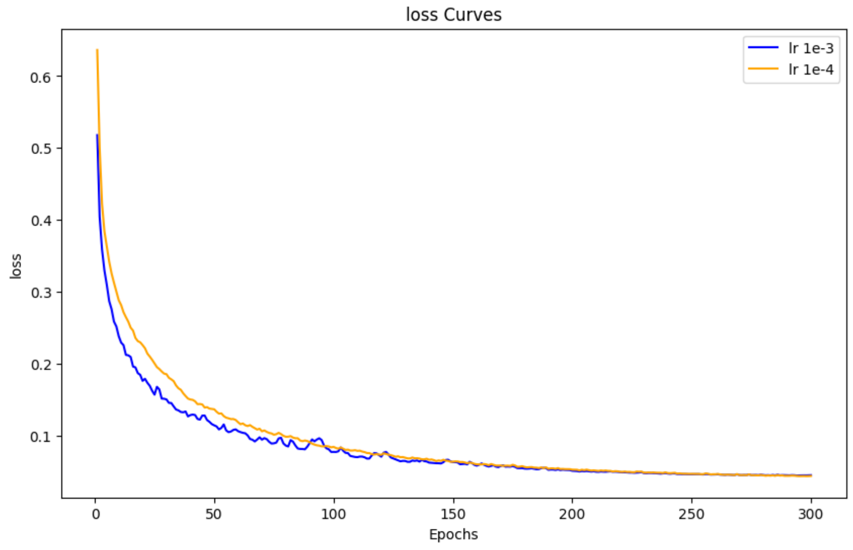


Figure 6: Resnet34 Learning Rate 1e-3 & 1e-4 Loss

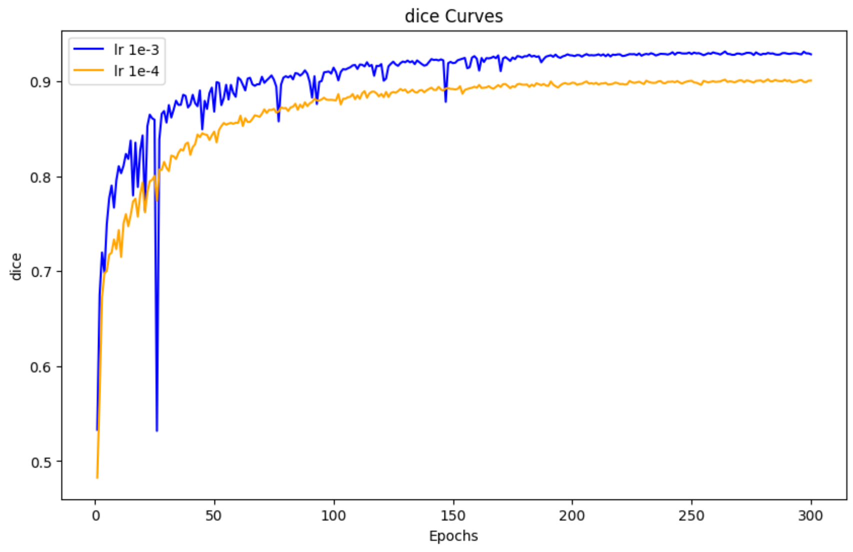


Figure 7: Resnet34 Learning Rate 1e-3 & 1e-4 Dice Score

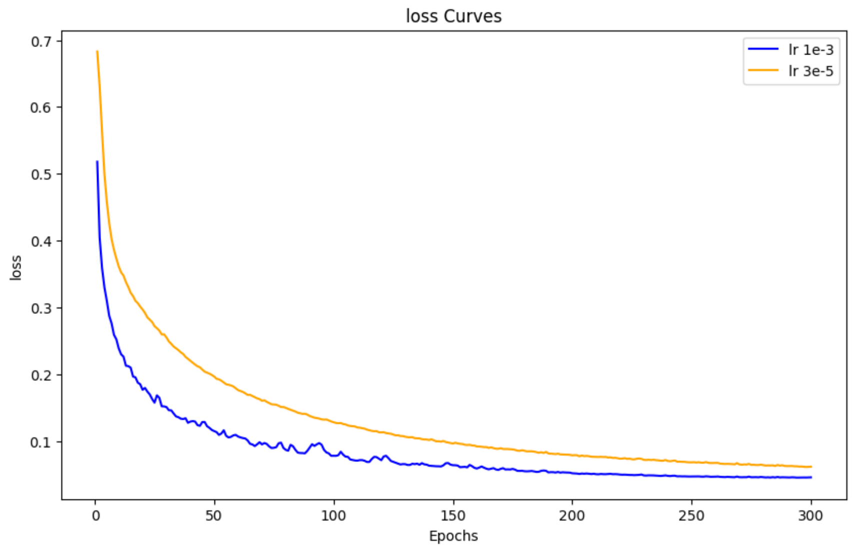


Figure 8: Resnet34 Learning Rate 1e-3 & 3e-5 Loss

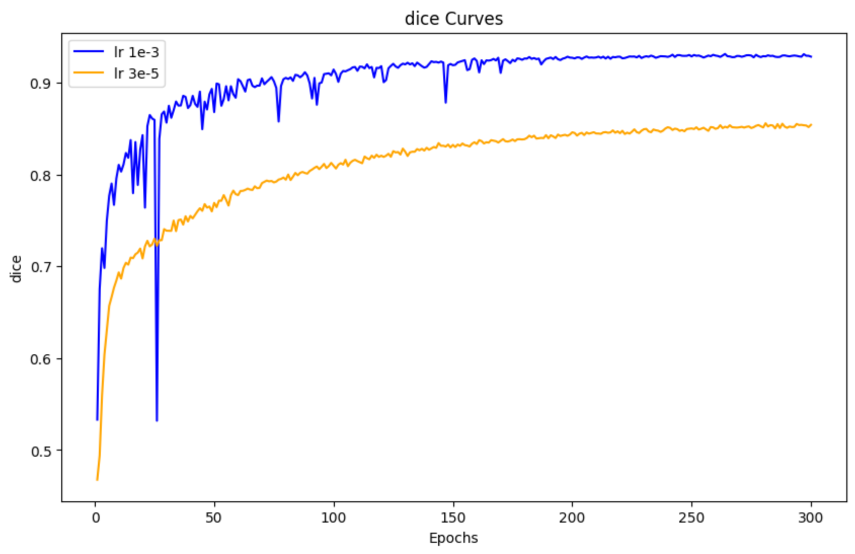


Figure 9: Resnet34 Learning Rate 1e-3 & 3e-5 Dice Score

In this task, I experimented with three common learning rates—1e-3, 1e-4, and 3e-5—using a CosineAnnealingLR scheduler, and eventually reduced the rate to 1e-6. The figures above show that a learning rate of 1e-3 achieved the best loss and Dice score compared to the other rates. Moreover, 1e-3 led to faster convergence, ultimately reaching a Dice score of 0.93. In my opinion, the reason 1e-3 works best is that the dataset is relatively simple, allowing the model to quickly learn its features. Therefore, a higher learning rate helps the model rapidly discover all the features needed to predict the mask. However, because the learning rate is higher, it may require more epochs to converge to a stable outcome, potentially increasing the overall training time.

3.3 Different Optimizers

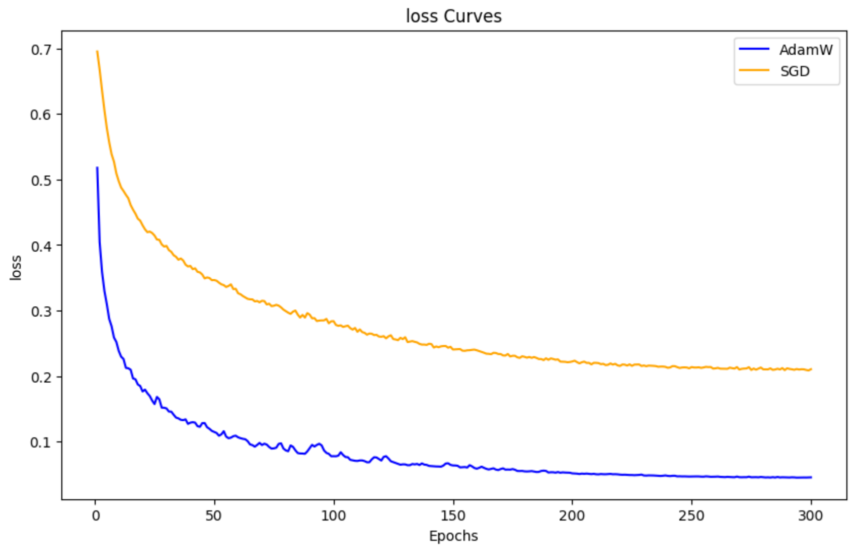


Figure 10: Resnet34 with AdamW & SGD Loss

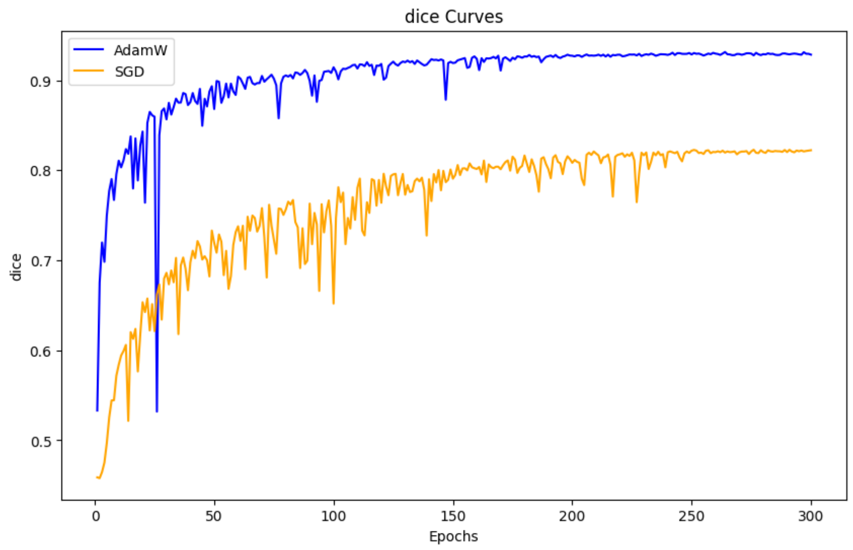


Figure 11: Resnet34 with AdamW & SGD Dice Score

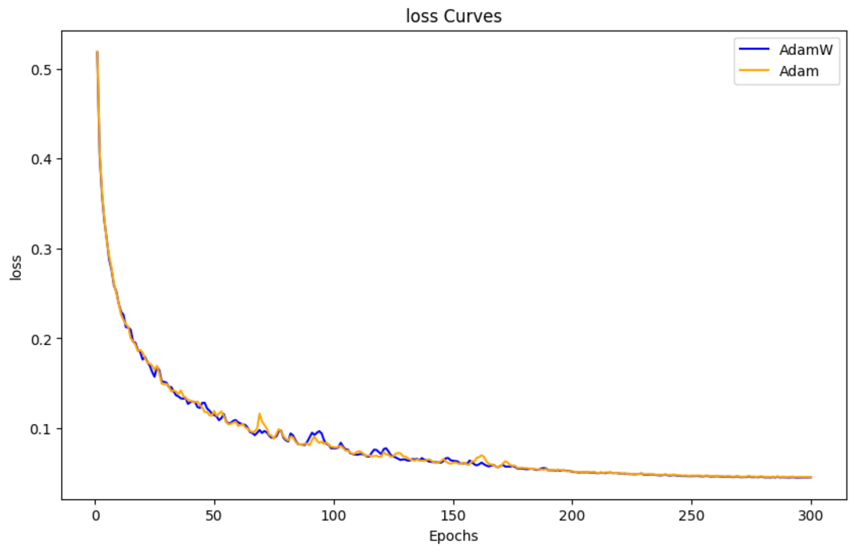


Figure 12: Resnet34 with AdamW & Adam Loss

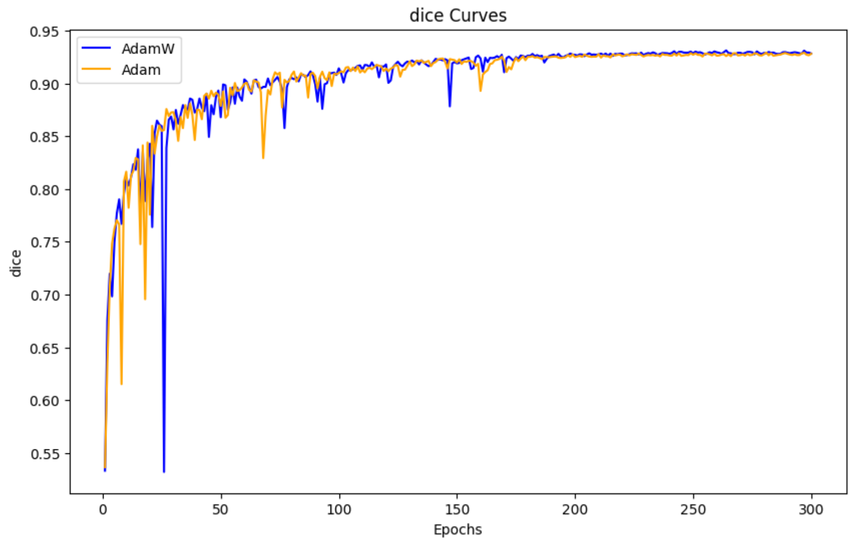


Figure 13: Resnet34 with AdamW & Adam Dice Score

For the optimizer part, I tried three optimizers—AdamW, Adam, and SGD—with the same hyperparameters for the rest of the training setup. The figures above show that SGD performed the worst, likely due to its sensitivity to other hyperparameters. In contrast, both Adam and AdamW achieved considerably higher Dice scores without any additional hyperparameter tuning, so I selected them as my primary choices.

The key difference between Adam and AdamW is that AdamW tends to be more unstable because of the way it incorporates regularization. This instability results in more significant weight updates, which helps prevent convergence to local minima and eventually leads to a better Dice score than Adam.

3.4 Different Loss Function

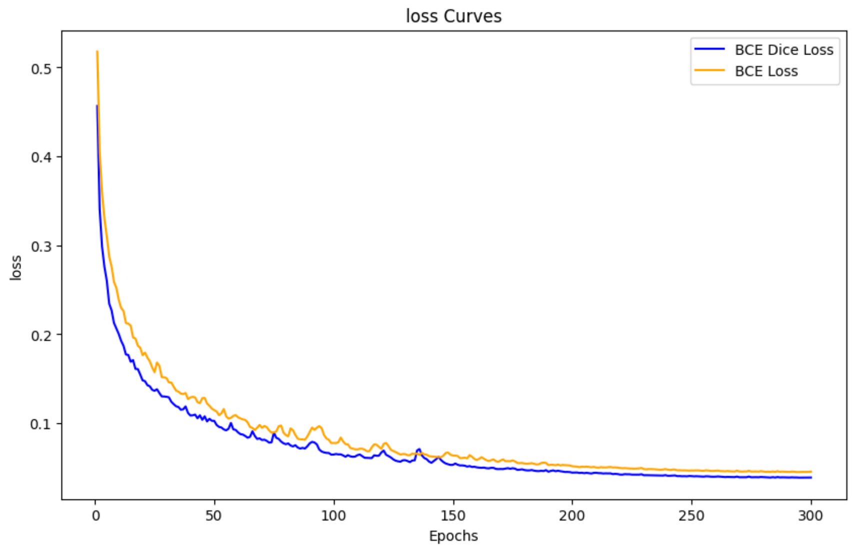


Figure 14: Resnet34 with BCE & DiceBCE Loss

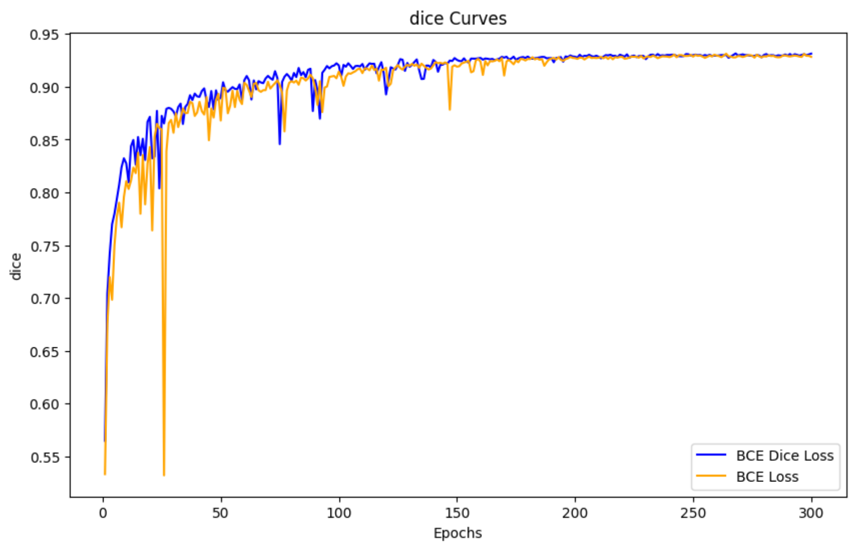


Figure 15: Resnet34 with BCE & DiceBCE Dice Score

For the loss function, I experimented with binary cross-entropy (BCE) loss and a customized loss function that combines BCE and Dice loss, each weighted at 0.5. Since this task is primarily evaluated using the Dice score, I believed that incorporating Dice loss into the training process would be beneficial. The figures above support this assumption, showing that the combined BCE and Dice loss achieves better overall performance than BCE alone.

3.5 Scheduler

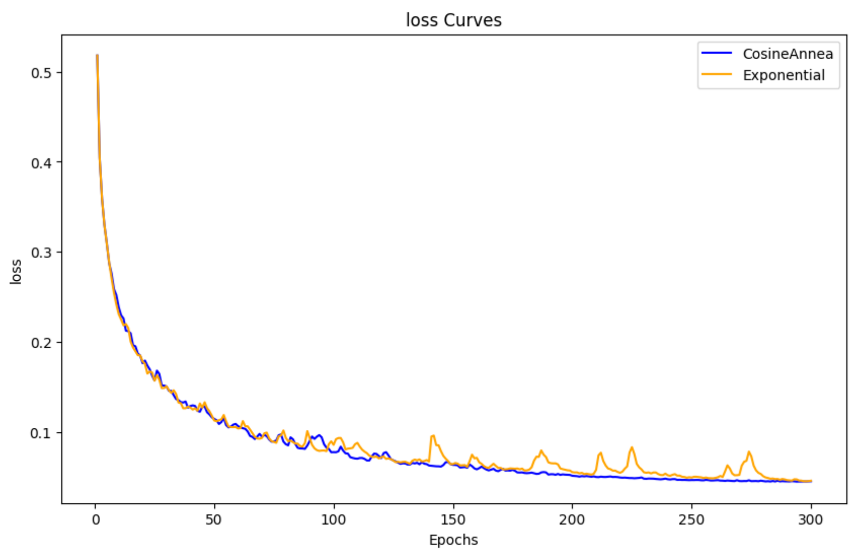


Figure 16: Resnet34 CosineAnnea vs. Exponential Loss

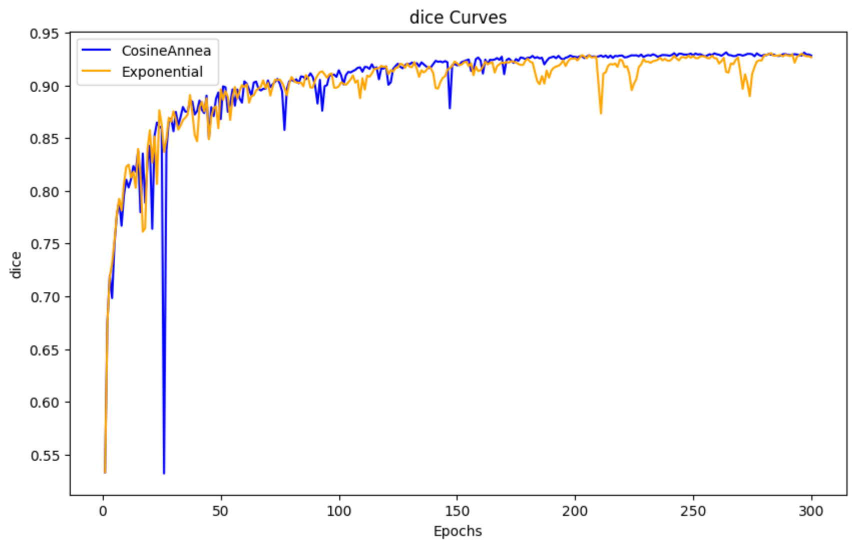
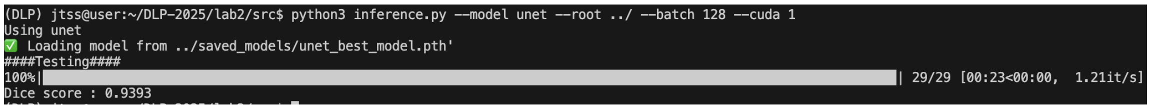


Figure 17: Resnet34 CosineAnnea vs. Exponential Dice Score

For the scheduler, I experimented with CosineAnnealingLR and ExponentialLR. The hyperparameters were set to decay the learning rate to 1e-5 by default for CosineAnnealingLR, while ExponentialLR used a decay factor of 0.999. Based on the experimental results, CosineAnnealingLR produced smoother loss curves over epochs, whereas ExponentialLR showed more fluctuations in the Dice score. Since Dice score is the primary evaluation metric for this task, and CosineAnnealingLR achieved a slightly higher Dice score, it was chosen as the preferred scheduler.

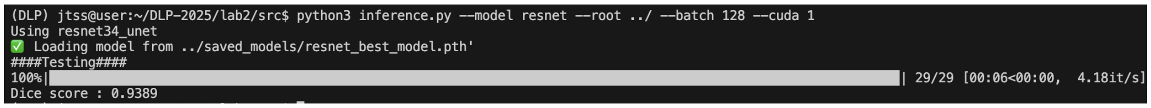
3.6 Training Conclusion

Based on multiple experiments, I have determined the best training strategy for this lab, achieving a result that meets the strong benchmark. During the experiments, I observed that the training set is very similar to the test set, meaning that a lower validation loss and higher Dice score generally lead to strong performance. However, if the test dataset changes, the results may not be as reliable, indicating that the implemented model's performance may not generalize as well to different datasets.



```
(DLP) jtss@user:~/DLP-2025/lab2/src$ python3 inference.py --model unet --root ../ --batch 128 --cuda 1
Using unet
✅ Loading model from ../saved_models/unet_best_model.pth'
####Testing####
100%|████████████████████████████████████████████████████████████████████| 29/29 [00:23<00:00,  1.21it/s]
Dice score : 0.9393
```

Figure 18: Unet Best Performance



```
(DLP) jtss@user:~/DLP-2025/lab2/src$ python3 inference.py --model resnet --root ../ --batch 128 --cuda 1
Using resnet34_unet
✅ Loading model from ../saved_models/resnet_best_model.pth'
####Testing####
100%|████████████████████████████████████████████████████████████████████| 29/29 [00:06<00:00,  4.18it/s]
Dice score : 0.9389
```

Figure 19: Resnet34 Best Performance

4 Execution steps: Training

4.1 Execution

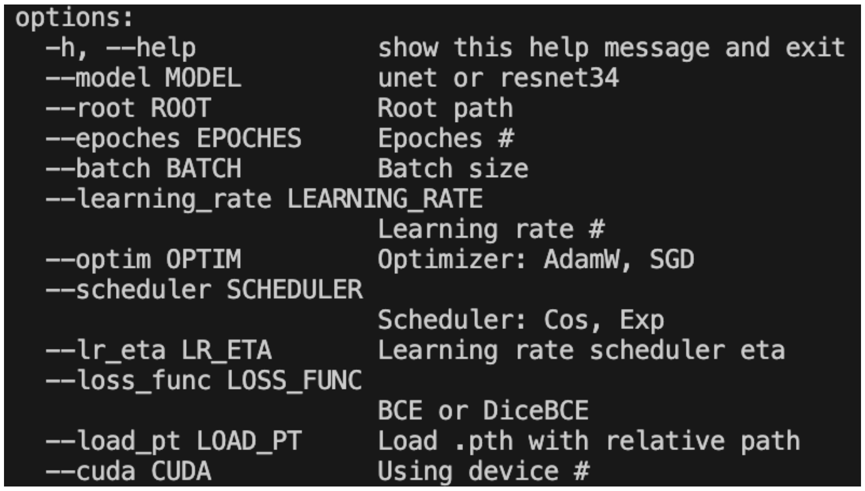
please execute under **./src** directory

./src:
Unet:
python3 train.py –root ../ –model unet –batch 128 –cuda 0 –epoches 300 –learning_rate 1e-3
Resnet34:
python3 train.py –root ../ –model resnet –batch 128 –cuda 0 –epoches 300 –learning_rate 1e-3

if others ie. ./lab2:
Unet:
python3 train.py –root . –model unet –batch 128 –cuda 0 –epoches 300 –learning_rate 1e-3
Resnet34:
python3 train.py –root . –model resnet –batch 128 –cuda 0 –epoches 300 –learning_rate 1e-3

please modify –root argument with relative directory to root directory against current directory.

4.2 Arguments



```
options:
  -h, --help            show this help message and exit
  --model MODEL         unet or resnet34
  --root ROOT           Root path
  --epoches EPOCHES     Epoches #
  --batch BATCH         Batch size
  --learning_rate LEARNING_RATE
                        Learning rate #
  --optim OPTIM         Optimizer: AdamW, SGD
  --scheduler SCHEDULER
                        Scheduler: Cos, Exp
  --lr_eta LR_ETA       Learning rate scheduler eta
  --loss_func LOSS_FUNC
                        BCE or DiceBCE
  --load_pt LOAD_PT     Load .pth with relative path
  --cuda CUDA           Using device #
```

Figure 20: Available Arguments

4.3 Default Settings

python3 train.py –model unet –root ../ –epoches 300 –batch 128 –learning_rate 1e-3 –optim AdamW –lr_eta 1e-5 –loss_func DiceBCE –load_pt ”” –cuda 0

To modify please follow options in the above subsection. And loading check point with –load_pt please type in the train .pth name.(not directory)

5 Execution steps: Inference

5.1 Execution

please execute under **./src** directory

./src:
Unet:
python3 inference.py –root ../ –model unet –batch 128 –cuda 0 –load_pt unet_best_model.pth
Resnet34:
python3 inference.py –root ../ –model resnet –batch 128 –cuda 0 –load_pt resnet_best_model.pth

if others (ie. ./lab2):
Unet:
python3 inference.py –root . –model unet –batch 128 –cuda 0 –load_pt unet_best_model.pth
Resnet34:
python3 inference.py –root . –model resnet –batch 128 –cuda 0 –load_pt resnet_best_model.pth

Please modify –root argument with relative directory to root directory against current directory. To load different .pth to inference, please just type the target .pth's file name instead of directory.

6 Discussion

6.1 Alternative Architectures

Attention mechanisms improve binary semantic segmentation in datasets like Oxford-IIIT Pet by helping the model focus on important features while reducing background distractions. In U-Net architectures, attention gates act as smart filters that highlight pet contours and key features while suppressing irrelevant background details, leading to more precise boundary detection.
Self-attention mechanisms help the model understand the relationship between different parts of the image by comparing all spatial positions. This is especially useful for distinguishing similar fur patterns, handling complex pet poses, and segmenting pets in difficult conditions like partial occlusions or poor lighting.
Channel attention modules, such as squeeze-and-excitation blocks, improve feature selection by emphasizing the most important channels while reducing the impact of less useful ones. This helps the model focus on breed-specific characteristics and ignore unnecessary background details. Spatial attention works alongside this by identifying key regions in the image, such as the ears, tail, and face, to enhance segmentation accuracy.
Combining both spatial and channel attention creates a powerful system that captures both fine textures and the overall shape of the pet. This hybrid approach improves segmentation accuracy, leading to higher Intersection over Union (IoU) scores and Dice coefficients. It is especially effective for handling pets with complex fur patterns or poses, producing clearer boundaries, fewer errors in textured backgrounds, and better preservation of important anatomical details.

This idea is inspired from https://github.com/bigmb/Unet-Segmentation-Pytorch-Nest-of-Unets.git which has multiple Unet-based architecture.

6.2 Potential Research

Future research in binary semantic segmentation for self-driving cars should focus on making models faster, more reliable, and better at handling different environments. Lightweight attention mechanisms and hybrid models can help improve real-time performance without requiring too much computing power. To handle challenges like rain, fog, or poor lighting, models could use weather-adaptive attention or combine data from different sensors like LiDAR and thermal cameras. Since labeling segmentation data is expensive, self-supervised and semi-supervised learning can help models learn from fewer labeled images. Adding uncertainty estimation methods can make models more aware of when they might be wrong, improving safety. Domain adaptation is also important so that models trained in one city work well in another without retraining. Using multiple sensor inputs, such as LiDAR, radar, and cameras together, can help detect objects more accurately, especially in difficult conditions. Finally, addressing biases in segmentation models is crucial to ensure fair and safe decision-making for all pedestrians and road users. Advancing these areas will make autonomous driving systems more accurate, adaptable, and safer in real-world conditions.